

알고리즘 프로젝트



과목명	컴퓨터알고리즘과실습
교수님	정 준 호 교수님
학과	컴퓨터공학과
조	6조


```
{ 0, 1, 2, 3, 4 } => 11111 => 31
{ 1, 2, 3, 4 }   => 11110 => 30
{ 1, 2, 4 }      => 10110 => 22
{ 2, 4 }         => 10100 => 20
{ 1 }            => 00010 => 2
```

즉, i번째 위치의 수를 추가 하고 싶다면 i번째 bit의 값을 1로 삭제하고 싶다면 0으로 변경하면 된다.
집합의 순서와 bit mask의 순서가 반대로 표현되는 이유: 2진수는 오른쪽에서 왼쪽으로 갈수록 커지기 때문이다.

AND 연산(&): 대응하는 두 bit가 모두 1일 때, 1을 반환 ex) 1010 & 1111 = 1010

OR 연산(|): 대응하는 두 bit 중 하나라도 1일 때 1을 반환 ex) 1010 | 1111 = 1111

시프트 연산(>>, <<): 왼쪽 또는 오른쪽으로 비트를 옮긴다.

ex) 00001010 << 2 = 00101000, 00001010 >> 2 = 00001000

< code >

```
#include <bits/stdc++.h>
#define xx first
#define yy second
using namespace std;
using pii = pair<int, int>;
using dpi = pair<pii, pii>;

int N, M;
int m[200][200];          //맵
bool vi[200][200][1 << 16]; //위치 + 청소할 칸의 정보를 bit masking으로 표현
int dy[4] = {-1, 1, 0, 0};
int dx[4] = {0, 0, -1, 1};

int bfs(int sty, int stx, int K) {
    // cout << "ck " << (1 << K) - 1;
    memset(vi, 0, sizeof(vi)); //배열을 0으로 초기화
    queue<dpi> q;               //큐 선언
    q.push({{0, 0}, {sty, stx}}); //queue에 0,0과 로봇의 위치 삽입
    vi[sty][stx][0] = false;    //로봇의 위치는 가지 않았다고 설정

    while (!q.empty()) {
        int cd = q.front().xx.xx; //queue의 첫번째, 이동한 횟수
        int ck = q.front().yy.yy; //queue의 두번째, 확인된 청소할 좌표 수
```

```

int cy = q.front().yy.xx; //queue의 세번째, 현재 y좌표
int cx = q.front().yy.yy; //queue의 네번째, 현재 x좌표
q.pop();
//총 4개라면 10000-1 => 1111
//cout << "ck " << ck << " " << ((1 << K) - 1) << " " << (1 << 16) << endl;
//모든 청소할 위치를 지났으면 ck의 상태는 1이 K개 반복된 2진수 형태
//ex) 청소할 좌표수가 4개 -> 1111
if (ck == (1 << K) - 1) {
    return cd; //몇 번 이동했는지 return
}
for (int i = 0; i < 4; i++) { // 4방향 이동
    int nd = cd + 1; //이동한 횟수 1증가
    int nk = ck; //다시 nk를 복구 -> 4방향 보기 위해서
    int ny = cy + dy[i]; //다음 이동할 위치
    int nx = cx + dx[i];
    //못 가는 곳이라면 continue
    if (ny < 0 || nx < 0 || ny >= N || nx >= M || m[ny][nx] == -1) continue;
    //다음 확인할 위치가 청소되었다면 or 연산을 통해 청소된 위치를 표시
    //ex)지금까지 1번째, 2번째 청소된 위치를 확인하고 3번째를 찾았다면
    //0011 | (1 << 3-1) = 0011 | (1 << 2) = 0011 | 0100 => 0111
    //즉, 0번째를 제외한 1, 2, 3 번째 청소된 위치를 확인
    if (m[ny][nx] > 0) nk |= (1 << (m[ny][nx] - 1));
    if (vi[ny][nx][nk]) continue; //이미 간 곳이라면 continue
    q.push({nd, nk}, {ny, nx}); //queue 에 push
    // cout << "nk " << nk << " " << nx << " " << ny << endl;
    vi[ny][nx][nk] = true; //간 곳이라고 표시해줌
}
}
return N * M - 1;
}

int main() {
    ios_base::sync_with_stdio(0);
    cin.tie(0);
    int tmp = 0;
    int cnt = 0;
    while (1) {
        cnt = 0;
        cin >> N >> M; //행, 열 수
        if (!N) break;
        memset(m, -1, sizeof(m)); //m 배열 초기화
        int sty, stx, K = 0; //로봇 위치, 청소된 위치 수
        string str;
    }
}

```

```

for (int i = 0; i < N; i++) {
    for (int j = 0; j < M; j++) {
        m[i][j] = tmp = rand() % 2; //랜덤 생성
        /* 청소된 좌표 갯수 리미트
        /if (tmp > 0) cnt++;
        if (cnt > 20) {
            m[i][j] = 0;
            tmp = 0;
        }*/
        cout << m[i][j] << " "; //방 상태 출력
        if ((tmp > 0) && (i != 0 || j != 0)) { //시작위치가 아니고 청소되었다면
            m[i][j] = ++K; //몇 번째로 확인했는지 입력
        }
    }
    cout << endl;
}
sty = 0, stx = 0, m[0][0] = 0; //로봇 위치 설정
cout << "출력 " << endl;
cout << bfs(sty, stx, K) << endl;
}
}

```

< code 설명 >

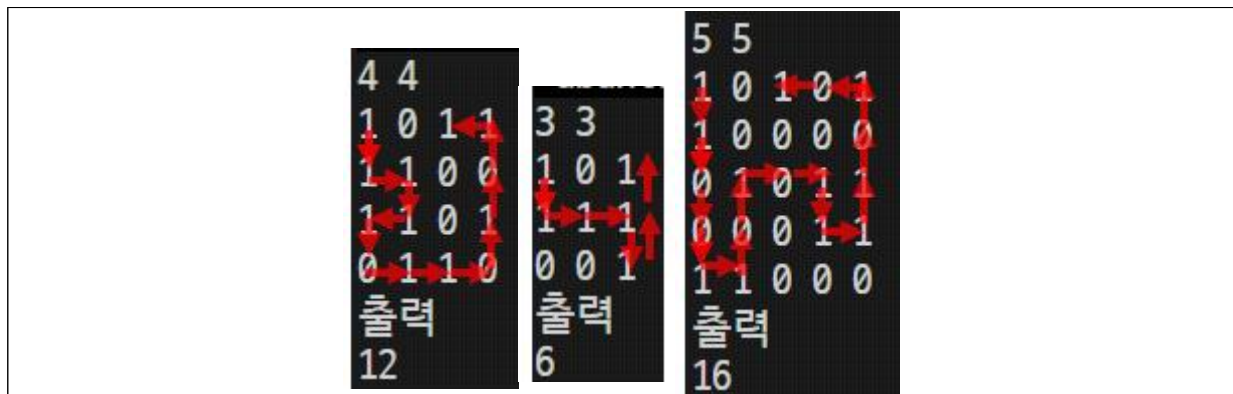
먼저 행과 열수를 입력받는다. 행이 0이라면 프로그램을 종료한다. 랜덤으로 0 또는 1의 값을 입력받아 지정된 좌표에 넣어준다. 1은 청소된 좌표 0은 청소되지 않은 좌표이다. 만약 청소된 좌표를 입력받았다면 몇 번째로 입력된 청소된 좌표인지 알려주는 K의 값을 증가시켜 해당하는 좌표에 값으로 넣어준다. 그 후 bfs 함수를 실행시킨다. 해당 좌표를 방문했는지 확인하는 vi 배열을 0으로 초기화해준다. 먼저 queue에 0,0과 로봇의 위치를 푸시해준다. cd는 현재까지 이동한 횟수를, ck는 확인한 청소된 좌표 수를, cy는 현재 y좌표를, cx는 현재 x좌표를 나타낸다. queue를 pop해준다. 만약 모든 청소된 위치를 지났다면, ck의 상태는 1이 K개 반복된 상태일 것이다.

예를 들어, 총 청소된 좌표 수가 4개라면 ck는 1111일 것이다. 즉, ck가 1을 4번 오른쪽으로 시프트 연산을 해준 후 1을 뺀 값 $10000 - 1 = 1111$ 과 같다면 모든 청소된 좌표를 찾은 것이므로 cd를 리턴해준다. 아직 찾지 못했다면 for문을 통해 4방향의 좌표를 확인한다. nd에는 cd에서 1을 더한 값을 넣어줘 이동한 횟수를 1 증가시켜준다. nk에는 ck값을 넣어준다. ny에는 다음 이동할 y의 좌표, nx에는 다음에 이동할 x의 좌표를 넣어준다. 만약 다음에 이동할 좌표가 가지 못하는 곳이라면 continue 해준다. 만약 다음에 이동할 위치가 청소된 곳이라면 or 연산을 통해 청소된 위치를 확인해준다.

예를 들어, 총 4개의 청소된 좌표가 있고 지금까지 1번째, 2번째 청소된 위치를 확인했고 이번

에 3번째를 찾았다면 nk는 0011의 상태일 것이다. nk는 $0011 \mid (1 \ll 3 - 1) = 0011 \mid (1 \ll 2) = 0011 \mid 0100 \Rightarrow 0111$ 이 된다. 즉, 0번째를 제외한 1, 2, 3번째 청소된 위치를 확인한 상태가 되는 것이다. 만약 다음에 갈 좌표가 이전에 갔던 적이 없다면 queue에 이동한 횟수, 확인한 청소된 좌표 수, x, y 값을 queue에 push 해준다. 만약 queue가 비워질 때까지 return 되지 못했다면 모든 좌표를 가야 한다는 뜻이므로, 행과 열을 곱한 수에서 1을 뺀 값을 return 해준다.

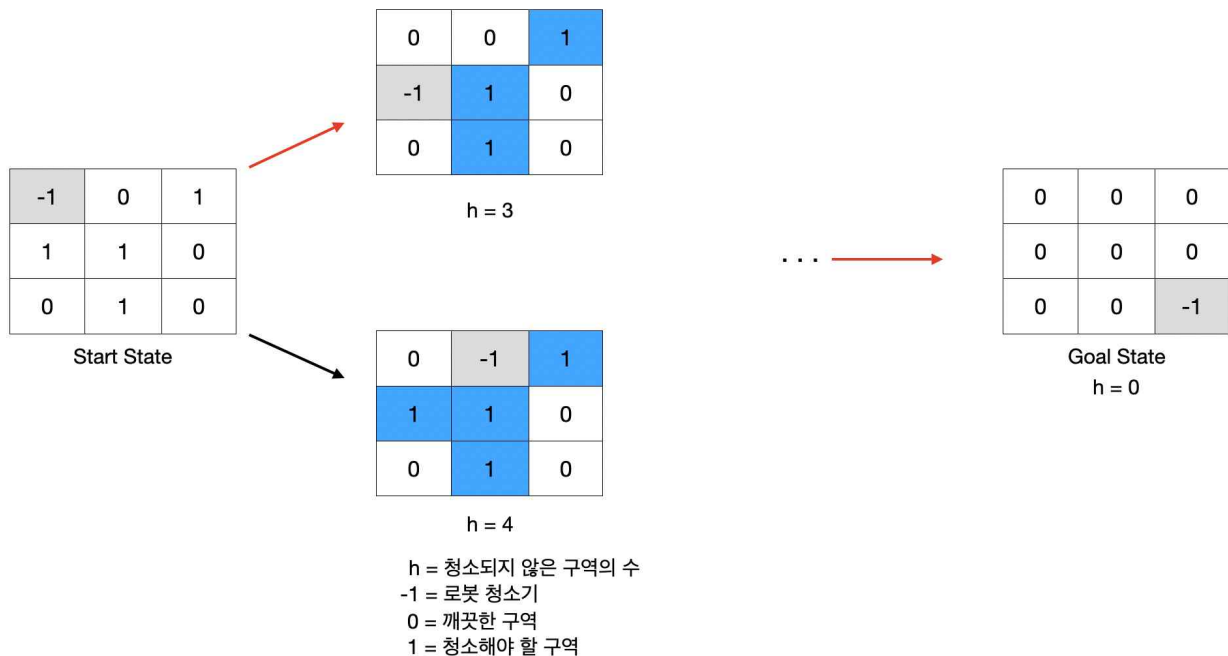
< 실행 결과 >



3-2) A*

< 알고리즘 설명 >

DFS와 BFS는 모든 상태 공간을 탐색해야하는 알고리즘이기 때문에 시간적, 공간적으로 매우 비효율적이다. 그래서 추정값을 기반으로 탐색하는 A* 알고리즘을 적용시켜보았다. A* 알고리즘의 특징은, 경로를 평가하기 위해 heuristic을 설정한다. heuristic이란, 다음 이동할 방향을 설정하기 위한 추정값이다.



예를 들어, 위 사진에서, 왼쪽의 Start state에서 오른쪽의 Goal State를 목표로 할 때 청소되지 않는 구역의 수를 heuristic으로 설정하였다. (0, 0)좌표에 위치한 청소기는, 두 방향(오른쪽, 아래쪽)으로 움직일 수 있다. 오른쪽으로 움직인 경우, h의 값은 4가 나오고, 아래쪽으로 움직인 경우, h의 값은 3이 나온다. 따라서, heuristic이 최소가 되는 아래쪽으로 청소기가 이동하게 된다.

이후, heuristic 값을 비교하며 Goal State에 도달할 때까지 반복한다. 이러한 방식을 사용하면 탐색 종료 시, 최적의 경로를 찾을 수 있다는 장점이 있다. 일반적인 경우에는 다른 Search 알고리즘인 BFS, DFS에 비해서는 상대적으로 시간복잡도가 좋지만, 최악의 경우 시간복잡도가 $O(b^d)$ (b: 확장되는 노드의 개수, d: 트리의 depth)로 여전히 비효율적이다.

< code >

```
#include <iostream>
#include <queue>
#include <string>
#include <cstdlib>
#include <ctime>

using namespace std;
int N, M; //방의 크기 N, M, 전역변수로 선언

struct Node {
    Node* parent; //부모 노드
    int** state; //노드 상태
    int x, y; //노드 좌표
    int g, h; //노드의 함수
```

```

string moves;          //노드의 방향

Node() {
    moves = "";
    state = new int* [N];
    for (int i = 0; i < N; i++)
        state[i] = new int[M];
}

};

//새로운 노드 생성 후 반환
Node* newNode(int** state, int x, int y, int dx, int dy, int g, Node* parent) {
    Node* node = new Node;
    node->parent = parent;

    for (int i = 0; i < N; i++) //새로운 노드에 기존 노드를 복사해준다.
        for (int j = 0; j < M; j++)
            node->state[i][j] = state[i][j];

    node->state[x][y] = 0;    //청소한 구역을 청소상태 0으로 만들어준다.
    node->state[dx][dy] = -1; //청소기가 이동한 곳에 -1(청소기)를 넣어준다.

    node->h = INT_MAX;
    node->g = g;

    node->x = dx;
    node->y = dy;
    return node;
}

//U, L, D, R
int dx[4] = { -1, 0, 1, 0 };
int dy[4] = { 0, -1, 0, 1 };

//최적의 이동경로 저장
string solutionPath = "";

//heuristic 함수 : heuristic은 state(map)에서 청소되지 않은 구역
//즉, 청소해야 할 구역의 개수를 세어 주었다.
int heuristic(int** state) {
    int h = 0;
    for (int i = 0; i < N; i++)

```



```

        for (int j = 0; j < M; j++)
            if (state[i][j] > 0)
                h++;
        return h;
    }

    //이동 가능 여부 판단 : map의 크기를 넘어서는 경우를 제외시켰다.
    bool isValid(int x, int y) {
        return (x >= 0 && x < N && y >= 0 && y < M);
    }

    //함수 계산 및 비교 : compare 함수는 node의 평가값을 비교하였다.
    struct compare {
        bool operator()(const Node* lhs, const Node* rhs) const {
            return (lhs->h + lhs->g) > (rhs->h + rhs->g);
        }
    };

    //ASTAR
    Node* ASTAR(int** startState, int x, int y) {
        //함수 평가값이 최소인 노드를 찾기 위해 우선순위 queue를 사용하여
        //평가값이 가장 작은 값을 우선시하였다.
        priority_queue<Node*, vector<Node*>, compare> open;

        Node* state = newNode(startState, x, y, x, y, 0, NULL);
        open.push(state);

        string move = "ULDR"; //이동 좌표 인덱스 활용

        while (!open.empty()) {
            Node* currentNode = open.top(); //평가값이 가장 작은 값을 가져온다.
            open.pop();

            if (currentNode->h == 0) //heuristic이 0인 경우 목표 노드 도달하고,
                return currentNode;

            //노드 expand, 현재 노드에서 위, 왼쪽, 아래, 오른쪽 순으로 평가값을 계산하여 우선순위 queue
            //에 삽입한다.
            for (int i = 0; i < 4; i++) {
                if (!isValid(currentNode->x + dx[i], currentNode->y + dy[i])) continue;

                Node* childNode = newNode( //현재 노드에서 이동할 노드(childNode)를 검사한다.
                    currentNode->state,

```

```

        currentNode->x,
        currentNode->y,
        currentNode->x + dx[i],
        currentNode->y + dy[i],
        currentNode->g + 1,
        currentNode);

    childNode->h = heuristic(childNode->state); //이동할 노드의 heuristic을 계산.
    childNode->moves = move[i]; //이동 방향을 할당
    open.push(childNode); //우선순위 queue에 push
    }
}
return NULL;
}

//노드 state 출력
void printState(int** state) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < M; j++)
            cout << state[i][j] << 'wt';
        cout << 'wn';
    }
}

//목표 노드에 도착하기까지의 과정 출력
void printSolutionPath(Node* node) {
    if (node == NULL) return;
    printSolutionPath(node->parent);
    solutionPath += node->moves + " - ";
    printState(node->state);
    cout << "wn";
}

int main() {
    srand(time(NULL));

    cout << "방의 크기 입력 : ";
    cin >> N >> M;

    //동적 할당
    int** startState = new int* [N];
    for (int i = 0; i < N; i++)
        startState[i] = new int[M];

```

```
//랜덤으로 방의 상태를 입력한다.
for (int i = 0; i < N; i++)
    for (int j = 0; j < M; j++)
        startState[i][j] = rand() % 2;

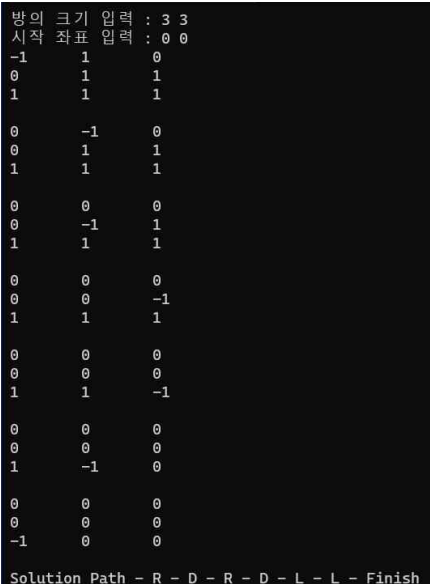
Node* goalNode = ASTAR(startState, 0, 0); //시작 state, 시작 x좌표, y좌표

if (goalNode == NULL) cout << "A* Failed \n";
else {
    printSolutionPath(goalNode);
    cout << "Solution Path" << solutionPath << "Finish\n";
}
}
```

< code 설명 >

방의 크기를 입력받은 후 이차원 배열에 방에 대한 상태를 깨끗한 구역인 0과 청소해야 할 구역인 1을 랜덤으로 생성해준다. A* 알고리즘을 사용하기 위해 노드를 구조체로 생성하여 부모의 노드, 노드의 상태, 노드의 좌표, 노드의 이동한 cost(g), node의 추정 cost(h), 노드가 이동하는 방향을 정의한다. heuristic은 청소해야 할 구역의 수로 정의한다. while 문을 사용하여, 각 노드에 대한 평가 값(g+h)를 계산하고, priority queue를 활용하여 평가값이 작은 수를 먼저 탐색한다. 노드의 heuristic이 0이 된다면 모두 청소했다고 판단하여 청소를 종료한다.

< 실행 결과 >



```
방의 크기 입력 : 3 3
시작 좌표 입력 : 0 0
-1 1 0
0 1 1
1 1 1

0 -1 0
0 1 1
1 1 1

0 0 0
0 -1 1
1 1 1

0 0 0
0 0 -1
1 1 1

0 0 0
0 0 0
1 1 -1

0 0 0
0 0 0
1 -1 0

0 0 0
0 0 0
-1 0 0

Solution Path - R - D - R - D - L - L - Finish
```

3-3) 기하

< 알고리즘 설명 >

시작 지점을 떠나 청소된 지점을 모두 한 번씩 방문하는 효율적인 경로를 생각했을 때, 청소기가 방문한 집들을 순서대로 직선으로 연결하고, 그 선분들이 서로 교차하지 않도록 하는 단순 폐쇄 경로 알고리즘을 이용했다. 3-1과 3-2에서는 Search 알고리즘들을 적용시켜 청소된 지점 사이의 최적 경로를 찾았지만, 시간복잡도가 매우 좋지 않았다. 따라서, 항상 최적의 경로라고는 할 수 없지만, 한번 통과한 지점을 다시 통과하지는 않으므로 어느 정도 효율적인 경로를 보장하고, 시간복잡도 측면에서도 장점이 있다. 또한, 앞의 알고리즘들은 무조건 상, 하, 좌, 우로만 움직여야하는 제한이 있었지만, 기하 알고리즘을 사용한다면 청소된 지점 사이를 직선 경로로 이동할 수 있는 장점이 있다.

< code >

```
#include <ctime>
#include <iostream>
#define MAXINT 101
using namespace std;

int N;
struct point {
    int x, y;    // x, y 값
    float angle; //수평각
    char c;
    point(int x = 0, int y = 0, char c = ' ') {
        this->x = x;
        this->y = y;
        this->c = c;
    }
};

struct line {
    struct point p1, p2;
};

void Swap(struct point* a, struct point* b) { //값 바꾸기
    struct point t;
    t = *a;
    *a = *b;
    *b = t;
}

float ComputeAngle(struct point p1, struct point p2) { //수평각 계산
    int dx, dy, ax, ay;    // x길이, y길이, x길이 절댓값, y길이 절댓값
    float t;                //각
```

```

dx = p2.x - p1.x;           // x길이
ax = abs(dx);               //절댓값
dy = p2.y - p1.y;           // y길이
ay = abs(dy);               //절댓값

t = (ax + ay == 0) ? 0 : (float)dy / (ax + ay); // 1사분면 or 같은 위치
if (dx < 0)
    t = 2 - t; // 2사분면
else if (dy <= 0) // 4사분면
    t = 4 + t;

return t * 90.0; //각 리턴
}

int partition(struct point a[], int l, int r) { //퀵정렬
    int i, j; //교환 할 수 찾고 바꾸기
    int v; //키 값
    if (r > l) {
        v = a[l].x; //제일 왼쪽 값을 키 값으로
        i = l; // i의 초기값을 제일 왼쪽 인덱스로 설정
        j = r + 1; // j의 초기값을 제일 오른쪽 인덱스에서 한칸 옆으로 설정
        for (;;) { //값이 범위를 벗어나지 않도록 함
            while (i + 1 < N && a[++i].x < v)
                ; //키 값보다 큰 값을 만날 때까지 i값 증가
            while (a[--j].x > v)
                ; //키 값보다 작은 값을 만날 때까지 j값 증가
            if (i >= j) //만약 i가 j보다 커지면 역전했으므로
                break; //나간다
            Swap(&a[i], &a[j]); // a[i]와 a[j]값을 바꿔준다
        }
    }
    Swap(&a[j], &a[l]); //키값과 a[j]값을 바꿔준다.
    return j; //다음 퀵정렬 기준이 되는 인덱스 리턴
}

void quicksort(struct point a[], int l, int r) { //퀵정렬 x좌표 기준으로 정렬
    int j;
    if (r > l) //아직 오른쪽 인덱스가 왼쪽 인덱스 보다 오른쪽에 있다면
    {
        j = partition(a, l, r); //교환할 수 찾기
        quicksort(a, l, j - 1); //구간 나눠서 퀵정렬 실행
        quicksort(a, j + 1, r);
    }
}

```

```

    }
}

void printMap(int** map, int row, int col) {
    for (int j = 0; j < row; j++) {
        for (int k = 0; k < col; k++) {
            cout << map[j][k] << 'Wt';
        }
        cout << endl;
    }
    cout << endl;
}

int main() {
    int miny = 100, checki;
    int x, y = 0, result, row, col, count = 1;
    char c;
    srand(time(NULL));
    cout << "방의 크기 입력 : ";
    cin >> row >> col;
    int** map = new int* [row];
    for (int i = 0; i < row; i++) {
        map[i] = new int[col];
    }

    N = col * row;
    struct point* polygon = new struct point[N + 2], z;
    checki = 0;

    for (int i = 0; i < row; i++) {
        for (int j = 0; j < col; j++) {
            map[i][j] = rand() % 2;
            cout << map[i][j] << "Wt";

            if (map[i][j] && !(i == 0 && j == 0)) {
                if (miny > j) {
                    miny = j;
                    checki = count;
                }
                else if (j == miny && polygon[checki].x > i) // y값이 같다면 x값이 가장 작은 점을
찾는다
                    checki = count;
                polygon[count++] = point(i, j, 'A');
            }
        }
    }
}

```

```

    }
}
cout << '\n';
}

for (int i = 1; i < count; i++) { //각각의 수평각 계산
    polygon[i].angle = ComputeAngle(polygon[checki], polygon[i]);
}

quicksort(polygon, 0, count - 1); //수평각을 기준으로 정렬

cout << endl;
for (int i = 0; i < count; i++) { //다각형 순서대로 출력

    map[polygon[i].x][polygon[i].y] = -1;
    printMap(map, row, col);
    map[polygon[i].x][polygon[i].y] = 0;
}
cout << "Solution Path";

for (int i = 0; i < count; i++) {
    cout << " - (" << polygon[i].x << ", " << polygon[i].y << ")";
}
}

```

< code 설명 >

방의 크기를 입력받은 후 이차원 배열에 방의 상태를 깨끗한 구역인 0과 청소해야 할 구역인 1을 랜덤으로 생성해준다. (0, 0) 좌표를 기준으로 1이 입력된 구역에 대해서 각각의 수평각을 계산한다. quick sort를 사용해 수평각을 오름차순 정렬하였다. 정렬한 좌표를 순서대로 이동하면 로봇청소기가 이동하는 최적의 경로가 된다.

< 실행 결과 >

```

방의 크기 입력 : 3 3
1   0   1
0   1   0
0   1   0

1   0   -1
0   1   0
0   1   0

-1  0   0
0   1   0
0   1   0

0   0   0
0   -1  0
0   1   0

0   0   0
0   0   0
0   -1  0

Solution Path - (0, 2) - (0, 0) - (1, 1) - (2, 1)

```